This reading describes alternative ways of expressing joins in SQL. We do not recommend using the techniques described in this reading, but you should familiarize yourself with them so you can read and understand SQL queries that use them.

SQL-92-Style Joins and SQL-89-Style Joins

In the video lectures describing joins in SQL, the following join syntax is used:

**SELECT ...**

**FROM toys JOIN makers**

**ON toys.maker\_id = makers.id;**

Notice the **JOIN**keyword between the table names, and the **ON**keyword followed by the join condition. This is called a *SQL-92-style join*, or *explicit join syntax*, and it is usually considered to be the best syntax to use for joins in SQL.

However, many SQL engines also support another join syntax, called the *SQL-89-style join*, or *implicit join syntax*. In this syntax, you use a comma-separated list of table names in the **FROM**clause, and you specify the join condition in the **WHERE**clause:

**SELECT ...**

**FROM toys, makers**

**WHERE toys.maker\_id = makers.id;**

With most SQL engines, this join query returns exactly the same result as the previous one.

With both join styles, you can use table aliases (**t**and **m**in this example):

**SELECT ...**

**FROM toys AS t JOIN makers AS m**

**ON t.maker\_id = m.id;**

**SELECT ...**

**FROM toys AS t, makers AS m**

**WHERE t.maker\_id = m.id;**

With both styles, the **AS**keyword before each table alias is optional.

When you use a SQL-89-style join, the SQL engine always performs an *inner*join. With this syntax, there is no way to specify any other type of join. If you want to use one of the other types of joins (left outer, right outer, full outer), then you must use a SQL-92-style join. Because of this limitation, and because the SQL-89-style join syntax makes it harder to understand the intent of the query, we recommend using SQL-92-style joins.

Unqualified Column References in Join Condition

In the join condition that comes after the **ON**keyword in a join query, the references to the corresponding columns are typically *qualified*with table names or table aliases. For example, when joining the toys table (alias **t**) and makers table (alias **m**), the join condition is specified as:

**ON t.maker\_id = m.id**

However, in the case where a bare column name unambiguously identifies a column, most SQL engines allow you to use a bare column name. For example, since there is no column named **maker\_id**in the makers table, the table alias **t**is not required in this join condition. So you could specify the join condition as:

**ON maker\_id = m.id**

But because there are columns named **id**in both tables, the table alias **m**is required in this join condition. If you omit the table alias **m**, then the SQL engine will throw an error indicating that the column reference **id**is ambiguous.

In join conditions, we recommend *always*qualifying column names with table names or table aliases, whether or not they are strictly required. Doing this makes your queries safer and clearer.

The USING Keyword

In some join queries, the names of the two corresponding columns in the join condition are identical. For example, in this query, the corresponding columns in the employees and offices table are both named **office\_id**:

**SELECT …**

**FROM employees e JOIN offices o**

**ON e.office\_id = o.office\_id;**

When the corresponding columns in the join condition have identical names, some SQL engines allow you to use a shorthand notation to specify the join condition. Instead of using the **ON**keyword and specifying the condition as an equality expression, you use the **USING**keyword and specify the common join key column name in parentheses after **USING**:

**SELECT …**

**FROM employees e JOIN offices o**

**USING (office\_id);**

Natural Joins

When the corresponding columns in the join condition have identical names, some SQL engines will allow you to omit the join condition, and will automatically join the tables on all the pairs of columns that have identical names in the left and right tables. To make a SQL engine do this, you need to specify the keyword **NATURAL**before the other join keywords. For example:

**SELECT …**

**FROM employees e NATURAL JOIN offices o;**

MySQL and PostgreSQL support natural joins, but Hive and Impala do not. In the SQL engines that support it, you can use the keyword **NATURAL**with any type of join; for example: **NATURAL LEFT OUTER JOIN**or **NATURAL INNER JOIN**.

Omitting Join Conditions

What happens if you attempt to perform a join without specifying the join condition, and you do *not*specify **NATURAL**before the join keywords?

For example, you might run a query like this:

**SELECT \***

**FROM toys JOIN makers;**

Notice that no join condition is specified. With some SQL engines (including PostgreSQL), this throws an error. But with other SQL engines (including Impala, Hive, and MySQL) this performs what’s called a *cross join*. In a cross join, the SQL engine iterates through each row in the table on the left side and combines it with every row in the table on the right side. So the result set includes every possible combination of the rows in the left table and the rows in the right table. The number of rows in the result set is the product (multiplication) of the number of rows in the left table and the number of rows in the right table (in this example, 3 x 3 = 9):

| **id** | **name** | **price** | **maker\_id** | **id** | **name** | **city** |
| --- | --- | --- | --- | --- | --- | --- |
| 21 | Lite-Brite | 14.47 | 105 | 105 | Hasbro | Pawtucket, RI |
| 21 | Lite-Brite | 14.47 | 105 | 106 | Ohio Art Company | Bryan, OH |
| 21 | Lite-Brite | 14.47 | 105 | 107 | Mattel | Segundo, CA |
| 22 | Mr. Potato Head | 11.50 | 105 | 105 | Hasbro | Pawtucket, RI |
| 22 | Mr. Potato Head | 11.50 | 105 | 106 | Ohio Art Company | Bryan, OH |
| 22 | Mr. Potato Head | 11.50 | 105 | 107 | Mattel | Segundo, CA |
| 23 | Etch A Sketch | 29.99 | 106 | 105 | Hasbro | Pawtucket, RI |
| 23 | Etch A Sketch | 29.99 | 106 | 106 | Ohio Art Company | Bryan, OH |
| 23 | Etch A Sketch | 29.99 | 106 | 107 | Mattel | Segundo, CA |

In most cases, the result of a cross join is meaningless. The rows of the result contain values with no correspondence. If you don’t realize that you have performed a cross join, you might be misled by the results. In addition, when performed on large tables, a cross join can return a dangerously large number of rows.

There are some specific cases when cross joins are useful, and in most SQL dialects, you can explicitly specify **CROSS JOIN**in your SQL statement to make it clear that you are performing a cross join. This is discussed in a video in the upcoming honors lesson.

So unless you intend to perform a cross join, and you understand the risks of this and how to interpret the output, we recommend specifying the join condition in every join query.

Explore

Top of Form

Bottom of Form

When you’re working as a data analyst, you will often need to use join queries to combine *three or more*tables. To do this, you use the same syntax as with two tables, but with more **JOIN**s added at the end of the **FROM**clause. Each **JOIN**should have its own **ON**keyword and join condition.

For example, to join the customers, orders, and employees tables, you could use this query:

**SELECT c.name AS customer\_name,**

 ​   ​   **o.total AS order\_total,**

​  ​    **e.first\_name AS employee\_name**

​   **FROM customers c**

​   **JOIN orders o ON c.cust\_id = o.cust\_id**

​   ​   **JOIN employees e ON o.empl\_id = e.empl\_id;**

Notice how the final two lines of this query have the same structure: the **JOIN**keyword, a table reference, a table alias, the **ON**keyword, and a join condition. You can add arbitrarily many lines like this to the **FROM**clause, to join together arbitrarily many tables.

The result the above query is:

| **customer\_name** | **order\_total** | **employee\_name** |
| --- | --- | --- |
| Arfa | 28.54 | Sabahattin |
| Brendon | 48.69 | Virginia |
| Brendon | -16.39 | Virginia |
| Chiyo | 24.78 | Ambrosio |

The arrangement of the rows in the result is arbitrary. Each result row represents an order, and gives the name of the customer who placed the order, the total amount of the order, and the employee who recorded the order. Because this is an inner join (the default type), all non-matching rows are excluded from the result. You can specify other types of joins, in any combination. For example, to include the order placed by the customer who is not in the customers table, change the first **JOIN**to **RIGHT OUTER JOIN**.

The sequence of the tables in a multi-table join does not matter, except with left and right outer joins, where it affects which table’s non-matching rows are included. Each join condition can refer to join key columns in any of the tables mentioned earlier in the **FROM**clause.

Join queries are computationally expensive and can be slow, especially when you’re joining three or more tables, or if you’re working with very large data. One strategy that can be used to remedy this problem is to join the data in advance and store the pre-joined result in a separate table, which you can then query. (If you completed the first course in this specialization, you might recall this is one way of *denormalizing*a normalized database, to make it easier to run analytic queries.) This approach of pre-joining tables has some costs, but it can make analytic queries easier to write and faster to run. A later course in this specialization will go into more details about how you can do that.